A Cache-Optimal Alternative to the Unidirectional Hierarchization Algorithm

Philipp Hupp and Riko Jacob

Abstract The sparse grid combination technique provides a framework to solve high-dimensional numerical problems with standard solvers by assembling a sparse grid from many coarse and anisotropic full grids called component grids. *Hierarchization* is one of the most fundamental tasks for sparse grids. It describes the transformation from the nodal basis to the hierarchical basis. In settings where the component grids have to be frequently combined and distributed in a massively parallel compute environment, hierarchization on component grids is relevant to minimize communication overhead.
We present a cache-oblivious hierarchization algorithm for component grids of the combination technique. It causes $|\mathbf{G}_{\boldsymbol{\ell}}| \cdot \left(\frac{1}{B} + \mathcal{O}\left(\frac{1}{\sqrt[d]{M}}\right)\right)$ cache misses under the tall cache assumption $M = \omega\left(B^d\right)$.[1] Here, $\mathbf{G}_{\boldsymbol{\ell}}$ denotes the component grid, d the dimension, M the size of the cache and B the cache line size. This algorithm decreases the leading term of the cache misses by a factor of d compared to the unidirectional algorithm which is the common standard up to now. The new algorithm is also optimal in the sense that the leading term of the cache misses is reduced to scanning complexity, i. e., every degree of freedom has to be touched once. We also present a variant of the algorithm that causes $|\mathbf{G}_{\boldsymbol{\ell}}| \cdot \left(\frac{2}{B} + \mathcal{O}\left(\frac{1}{\sqrt[d-1]{M \cdot B^{d-2}}}\right)\right)$ cache misses under the assumption $M = \omega(B)$. The new algorithms have been implemented and outperform previously existing software. In several cases the measured performance is close to the best possible.

Key words: sparse grids, combination technique, hierarchization, cache misses, external memory, cache-oblivious algorithms, unidirectional principle

Philipp Hupp
ETH Zürich

Riko Jacob
IT University of Copenhagen, Rued Langgaards Vej 7, DK-2300 København S, Denmark e-mail: rikj@itu.dk

[1] The dimension d is assumed to be constant in the $\mathcal{O}$-notation.

1 Introduction

The gap between peak performance and memory bandwidth on modern processors is already large and still increasing. In many situations, this phenomenon can be counteracted by using caches, i.e., a small but fast additional memory close to the processor. Now, the expensive communication is between the memory and the cache, and this kind of communication efficiency is crucial for high performance code. All areas of computer science acknowledge this phenomenon, but call it and the methods to design such algorithms slightly differently. The algorithms that reduce memory traffic are called, e. g., I/O efficient algorithms [22, 1, 31], communication avoiding algorithms [27, 2, 13], and blocked algorithms [30]. Still, all these efforts aim to increase temporal locality (reuse over time) and spatial locality (use of several items of a cache line) to reduce the amount of data that is transfered between the different levels of the memory hierarchy.

Sparse grids [40, 41, 3] are a numerical discretization scheme that allows to solve high-dimensional numerical problems by lessening the curse of dimensionality from $\mathcal{O}\left(h_n^{-d}\right)$ to $\mathcal{O}\left(h_n^{-1} \cdot |\log_2 h_n|^{d-1}\right)$ for dimension d and minimum mesh size $h_n = 2^{-n}$. Crucial for the reduction in the degrees of freedom is a change of basis from the nodal basis to the hierarchical basis and the selection of the most important basis functions of the hierarchical basis. This change of basis is called *hierarchization* and is one of the most fundamental algorithms for sparse grids. The reduction in the degrees of freedom for sparse grids comes at the cost of a less regular structure and more complicated data access patterns for sparse grid algorithms. In consequence, communication efficient algorithms are in particular important and less obvious for sparse grids. Because hierarchization is among the most simple algorithmic tasks that are based on the hierarchical structure of the sparse grids, we consider it prototypical in the sense that algorithmic ideas that work for it are also applicable to more complicated tasks.

The sparse grid combination technique [17] assembles the sparse grid from a linear combination of many coarse and anisotropic, i. e. refined differently in different dimensions, full grids called component grids. This allows to solve the numerical problem on the full component grids with standard solvers while taking advantage of the reduced number of degrees of freedom of the sparse grid. For time dependent problems, the combination technique can be applied as depicted in Figure 1: a standard solver is employed to each of the (regular) component grids. Then, a reduce step assembles the sparse grid solution as a linear combination of the component grid solutions. This is followed by a broadcast step that distributes the joint solution back to the component grids. The change of basis from the regular grid basis to the hierarchical basis can facilitate the reduce and the broadcast step. In this situation, hierarchization is on the performance critical path of the solver. Current approaches to master large simulations of hot fusion plasmas are a prominent example [37].

The task of hierarchization we consider here has as input an array of values representing the sampled function in the nodal basis, i.e., as function values sampled at the grid point, and as output the same function represented in the hierarchical basis

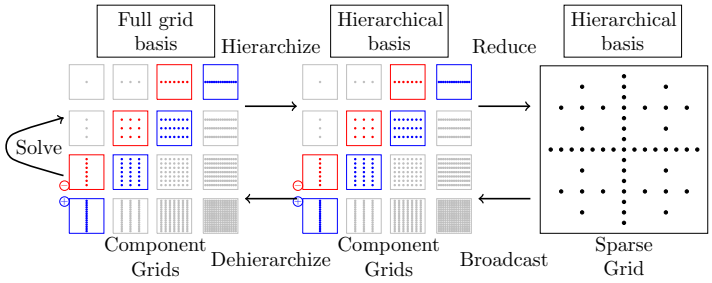

Fig. 1 (De-)hierarchization as pre- and postprocessing steps for the reduce/broadcast step of the combination technique. The combination technique computes a solution in the sparse grid space by a suitable linear combination (blue: $+1$; red: -1) of the component grid solutions.

as explained later. The grid points form a regular anisotropic (not all dimensions are refined equally) grid, and the coefficient values are laid out in lexicographic order, i.e., in a generalized row major layout. The algorithm is formulated and analyzed in the cache oblivious model [10]. In this model the algorithm is formulated to work on a random access memory where every memory position holds an element, i.e., an input value, an intermediate value or an output value. It is analyzed in the I/O model with a fast memory (cache) that holds M elements, the transfer to the slower memory is done in blocks or cache lines with B elements, and the cache replacement strategy is optimal. This makes sure that elements that participate in algebraic operations reside in cache. The performance is measured as the number of cache misses (also called I/Os) the algorithm incurs. Note that in this model the CPU operations are not counted as if the CPU was infinitely fast. Hence, this model focuses on one level of the memory hierarchy, usually the biggest relevant one. As our experiments demonstrate, these assumptions capture the most important aspects of our test machine. More precisely, it turns out that the shared level 3 cache of the CPU is usually the bottleneck, because the four cores of the CPU together are fast enough to keep the memory connection busy all the time. Hence it is reasonable as a theoretical model to regard the CPU(s) as infinitely fast.

The unidirectional principle is the dominating design pattern for sparse grid algorithms. The unidirectional principle exploits the tensor product structure of the underlying basis and decomposes the global operator into d sweeps over the grid. In each sweep it works locally on all one-dimensional subproblems, called poles, of the current work direction [3]. By working in d sweeps the unidirectional hierarchization algorithm (Algorithm 1) needs only $3d$ arithmetic operations to hierarchize piecewise d-linear basis functions. In contrast, a direct hierarchization algorithm, i. e., calculating the hierarchical surplus (the coefficient in the hierarchical basis) of each grid point in one go, like formulating the task as a multiplication with a sparse matrix, would require $c \cdot 3^d$ arithmetic operations for $1 \leq c \leq 2$. Therefore, the unidirectional algorithm is a good choice with respect to the number of arithmetic operations. As such, the unidirectional algorithm is, however, inherently cache inefficient in the sense that it performs d sweeps over the data and therefore causes at

least $d \cdot \frac{|\mathbf{G}_\ell|}{B} - (d-1) \cdot \frac{M}{B} = d \cdot \frac{1}{B} \cdot \left(|\mathbf{G}_\ell| - \frac{d-1}{d} \cdot M \right)$ cache misses. Here, $\mathbf{G}_\ell$ denotes the input grid, M the size of the internal memory or cache and B the block or cache line size. Furthermore, the unidirectional hierarchization algorithm has been implemented for component grids such that it is within a factor of 1.5 of this unidirectional memory bound [24]. In consequence, any significant further improvements have to avoid the unidirectional principle on a global scale.

This paper presents a cache-oblivious [10] hierarchization algorithm (Algorithm 2) that avoids the unidirectional principle on a global scale but applies it (recursively) to smaller subproblems that fit into cache. It actually computes precisely the same intermediate values at the same memory locations as the unidirectional algorithm, but it computes them in a different order. By doing so, the algorithm avoids the d global passes of the unidirectional algorithm. For component grids and the piecewise-linear basis this algorithm causes $|\mathbf{G}_\ell| \cdot \left(\frac{1}{B} + \mathcal{O}\left(\frac{1}{\sqrt[d]{M}} \right) \right)$ cache misses, i. e., it works with scanning complexity (touching every grid point once) plus a lower order term. For the second term to be of lower order a strong tall cache assumption of $M = \omega\left(B^d\right)$ is needed. It reflects that we, as is usual (e.g. [10, 38]), consider the asymptotics of increasing M, and here in particular demand that M grows faster than B^d. With this strong tall cache assumption, the leading term of this complexity result is optimal, as every algorithm needs to scan the input. In addition, the presented algorithm reduces the leading term of the cache misses by at least a factor of d compared to any unidirectional algorithm. For the situation that the cache is not that tall but only of size $M = \omega(B)$, we give a variant of the algorithm that causes at most $|\mathbf{G}_\ell| \cdot \left(\frac{2}{B} + \mathcal{O}\left(\frac{1}{\sqrt[d-1]{M \cdot B^{d-2}}} \right) \right)$ cache misses. Depending on the size of the cache, the leading term of the cache misses is therefore reduced by a factor of d or $d/2$ compared to the unidirectional algorithm. The presented algorithm is cache-oblivious, works on a standard row major layout, relies on a least recently used (LRU) cache replacement strategy, is in-place, performs the same arithmetic operations as the unidirectional algorithm and works for anisotropic component grids.

To ease readability and in agreement with common usage in the sparse grid literature, this paper generally assumes for the $\mathcal{O}$-notation that the dimension d is constant: In numerics, the dimension d is a parameter inherent to the problem under consideration. If a more accurate solution is required, the refinement level of the discretization is increased while the dimension of the problem stays constant. For completeness, we state the complexity of the divide and conquer hierarchization algorithm for component grids also including the constant d at the end of the relevant section.

The rest of the paper is organized as follows: Section 2 considers related work, Section 3 explains the relevant concepts of sparse grids and how they are presented in this paper, Section 4 formulates the algorithm and analyses it, Section 5 reports on run time experiments of an implementation on current hardware, and Section 6 discusses conclusions and directions for future work.

2 Related Work

Hong and Kung started the analysis of the I/O-complexity of algorithms with their red-blue pebble game [22] assuming an internal memory of size M and basic blocks of size $B = 1$.[2] To use data for computations, it has to reside in internal memory. Aggarwal and Vitter extended this basic model to the cache-aware external memory model [1] with arbitrary block or cache line size B to account for spatial locality. Frigo et. al. generalized this to the cache-oblivious model [10] in which the parameters M and B are not known to the algorithm and the cache replacement strategy is assumed to be the best possible. As the parameters M and B are not known when a cache-oblivious algorithm is designed, a cache-oblivious algorithm is automatically efficient for several layers of the memory hierarchy simultaneously. All mentioned theoretical models assume a fully associative cache and so does the analysis presented in this paper.

Sparse grids [40, 41, 3] and the sparse grid combination technique [17] have been developed to solve high-dimensional numerical problems. They have been applied to a variety of high-dimensional numerical problems, including partial differential equations (PDEs) from fluid mechanics [16], financial mathematics [21, 4] and physics [29], real-time visualization applications [7, 6], machine learning problems [12, 11, 36], data mining problems [12, 5] and so forth. In a current project [37], the combination technique is used as depicted in Figure 1 to simulate hot fusion plasmas as they occur in plasma fusion reactors like the international flagship project ITER. In this project the fusion plasmas are modeled using the gyrokinetic approach which results in a high-dimensional PDE, i. e., five space and velocity dimensions plus time. Furthermore, the convergence of the combination technique has been studied for several special cases [34, 35, 39] as well as general operator equations [14].

Due to the coarse grain parallelism of the component grids the combination technique is ideal for high performance computing [19]. This coarse grain parallelism also allows to incorporate algorithm based fault tolerance into the combination technique [20]. It was discovered early that, for time dependent PDEs, the component grid solutions need to be synchronized after few time steps [15] and that the communication needed for this synchronization can be reduced if the component grid solutions are represented in the hierarchical basis [18]. Recently, communication schemes that use the hierarchical representation of the component grid solutions to minimize communication in this synchronization step were derived, implemented and tested for the setting of the gyrokinetic approach [26, 23, 25].

The problem considered in this work, namely finding efficient hierarchization and dehierarchization algorithms for sparse grids, has been investigated in many occasions [9, 36, 32, 33, 28, 6, 24, 8]. All these algorithms implement the unidirectional algorithm and hence sweep d times over the whole data set. The unidirectional hierarchization algorithm for component grids has been implemented such that it is within a factor of 1.5 of the unidirectional memory bound [24]. Therefore, any

[2] We use the terms internal memory and cache as well as cache line size and block size synonymously.

significant further improvements have to avoid the global unidirectional principle. This paper extends the first algorithm that avoids the unidirectional principle on a global scale [23]. In contrast to this initial version of the algorithm, the algorithm presented in this paper works in-place and performs the same arithmetic operations as the unidirectional algorithm. Also, the first lower bound for the hierarchization task was proven in [23].

3 Sparse Grid Definitions and the Unidirectional Hierarchization Algorithm

This section describes the necessary notation and background to discuss the sparse grid hierarchization algorithm. For a thorough description of sparse grids we refer to the survey by Bungartz and Griebel [3].

Let us begin with a conventional level ℓ discretization of the 1-dimensional space $\Omega := [0,1]$. The grid points x of $\mathbf{G}_\ell$ are

$$\mathbf{G}_\ell = \left\{ x = \frac{i}{2^\ell} \in \Omega : \; i \in \left\{0,1,\ldots,2^\ell\right\} \right\} .$$

The corresponding nodal basis functions are the piecewise linear hat functions with peak at the grid point and support of the form $]\frac{i-1}{2^\ell}, \frac{i+1}{2^\ell}[$. The 1-dimensional sparse grid of level $n = \ell$ has the same grid points. We use the notation $x_{k,i} = \frac{i}{2^k}$ for $0 \le k \le \ell$ and $i \in \{0,\ldots,2^k\}$. Two distinct pairs (k,i) and (k',i') describe the same grid point if the coordinates of the grid points are identical, i. e., if $x_{k,i} = x_{k',i'}$. For a level-index pair (k,i) the reduced pair (k',i') is defined to have the smallest i', i. e., i' is odd, with $x_{k',i'} = x_{k,i}$. For odd i and $1 \le k$, the interval $I_{k,i} =]x_{k,i} - 2^{-k}, x_{k,i} + 2^{-k}[$, is the support of the corresponding hierarchical basis function $\phi_{k,i}(x)$ with $x_{k,i}$ as its midpoint, i.e. $\phi_{k,i}(x) = \max(0, 1 - |x - x_{k,i}| \cdot 2^k)$ as is also depicted in Figure 2. The reduced level index pairs of the two endpoints (allowing $(0,0)$ as a special case) define the functions $\mathscr{L}$ and $\mathscr{R}$ by $I_{k,i} =]x_{\mathscr{L}(k,i)}, x_{\mathscr{R}(k,i)}[$. The two grid points $x_{\mathscr{L}(k,i)}$ and $x_{\mathscr{R}(k,i)}$ are called the left and respectively right hierarchical predecessor. We additionally define the interval $I_{0,0} = [0,1]$. $I_{0,0}$ differs from $I_{1,1} =]0,1[$ only by its two endpoints which are called global boundary points. These two global boundary points are the only grid points that have no hierarchical predecessors. The closure $\overline{I_{k,i}}$ of an interval $I_{k,i} =]x_{\mathscr{L}(k,i)}, x_{\mathscr{R}(k,i)}[$ is defined in the usual way as $\overline{I_{k,i}} := [x_{\mathscr{L}(k,i)}, x_{\mathscr{R}(k,i)}]$.

We say that $I_{k,i}$ has the two children intervals $I_{k+1,2i-1} =]x_{\mathscr{L}(k,i)}, x_{k,i}[$ and $I_{k+1,2i+1} =]x_{k,i}, x_{\mathscr{R}(k,i)}[$. We extend this notion (by transitive closure) to descendants, and observe that an interval $I_{k',i'}$ is a descendant of $I_{k,i}$ if and only if $I_{k',i'} \subset I_{k,i}$. This immediately leads to the following statement:

Lemma 1. *If a grid point $x_{k',i'}$ is element of the interval $I_{k,i}$ (with odd i and i') then $k' \ge k$ and the hierarchical predecessors of $x_{k',i'}$ are in the closure of $I_{k,i}$, i. e., $\left(x_{k',i'} \in I_{k,i}\right) \Rightarrow \left(\left\{x_{\mathscr{L}(k',i')}, x_{\mathscr{R}(k',i')}\right\} \subset \overline{I_{k,i}}\right)$.*

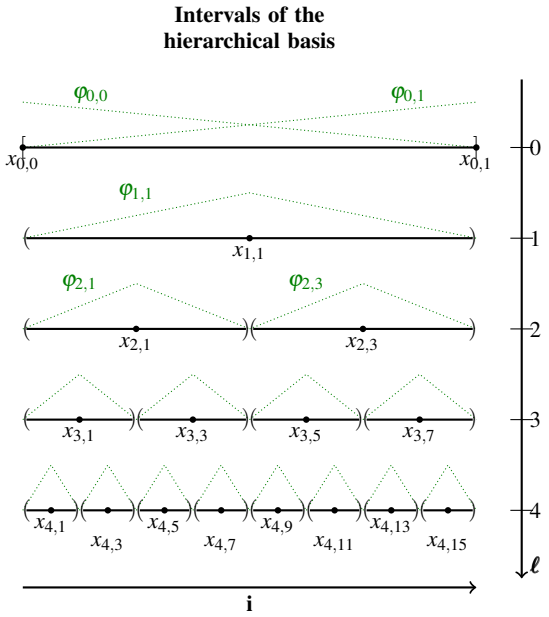

Fig. 2 The intervals $I_{\mathbf{k},\mathbf{i}}$ and the corresponding grid points $\mathbf{x}_{\mathbf{k},\mathbf{i}}$ as well as the piecewise linear basis functions $\varphi_{\mathbf{k},\mathbf{i}}$ of the hierarchical basis.

Hierarchization is a change of basis of a piecewise linear function from the nodal basis of level ℓ, given as the function values y_i at position $i2^{-\ell} \in \{0, 2^{-\ell}, \ldots, 1\}$, into the hierarchical basis of the sparse grid. For odd i, $k \leq \ell$ and $I_{k,i} =]x_{\mathscr{L}(k,i)}, x_{\mathscr{R}(k,i)}[$, we get the hierarchical surplus as $\alpha_{k,i} = y_{k,i} - \frac{1}{2}(y_{\mathscr{L}(k,i)} + y_{\mathscr{R}(k,i)})$ for $k \geq 1$, and for the global boundary points we have $\alpha_{0,0} = y_{0,0}$ and $\alpha_{0,1} = y_{0,1}$. The grid points $\{x_{k,i}, x_{\mathscr{L}(k,i)}, x_{\mathscr{R}(k,i)}\}$ form the 3-point stencil of $x_{k,i}$.

Let us now address the d-dimensional case and the discretization of the space $\Omega := [0,1]^d$. In general, vectors are written in bold face and operations on them are meant component wise. The conventional anisotropic grid $\mathbf{G}_{\boldsymbol{\ell}}$ with mesh-width $h_{\boldsymbol{\ell}_r} := 2^{-\boldsymbol{\ell}_r}$ and discretization level $\boldsymbol{\ell}_r$ in dimension $r \in \{1, \ldots, d\}$ has the grid points

$$\mathbf{G}_{\boldsymbol{\ell}} = \left\{ \mathbf{x} = \frac{\mathbf{i}}{2^{\boldsymbol{\ell}}} \in \Omega : \; i_r \in \left\{ 0, 1, \ldots, 2^{\boldsymbol{\ell}_r} \right\} \forall r \in \{1, \ldots, d\} \right\} .$$

The corresponding basis functions are the tensor products of the one-dimensional basis functions. Hence, the grid $\mathbf{G}_{\boldsymbol{\ell}} \subset [0,1]^d$ is completely defined by its level vector $\boldsymbol{\ell} \in \mathbb{N}_0^d$ describing how often dimension $r \in \{1, \ldots, d\}$ has been refined. A grid of refinement level ℓ_r consists of $2^{\ell_r} + 1$ grid points in dimension r, the outermost two of which are called global boundary points, i. e., the points with $i_r \in \left\{0, 2^{\ell_r}\right\}$.

The d-dimensional sparse grid results from a tensor product approach. To express this, a level and index is replaced by a d-fold level- and index-vector in the above

definition of the sparse grid. The grid points have the form $(x_{k_1,i_1},\dots,x_{k_d,i_d})$ corresponding to the basis function with support $I_{k_1,i_1}\times\cdots\times I_{k_d,i_d}$. The regular sparse grid of level n consists of the grid points with $|\mathbf{k}|_1 \le n+d-1$, i.e., $k_1+\cdots+k_d \le n+d-1$. The anisotropic component grid with level vector $\boldsymbol{\ell}$ consists of the grid points with $k_r \le \boldsymbol{\ell}_r$.

In this case hierarchization can be performed using the unidirectional principle using $d-1$ intermediate results at every grid point. More precisely, we define $d+1$ variables $\alpha^{(j)}_{\boldsymbol{\ell},\mathbf{i}}$. For $j=0$, the variable $\alpha^{(0)}_{\boldsymbol{\ell},\mathbf{i}}$ is the function value at position described by $(\mathbf{i},\boldsymbol{\ell})$. The final value $\alpha^{(d)}_{\boldsymbol{\ell},\mathbf{i}}$ is the hierarchical surpluses, i.e., the coefficient of the hierarchical basis functions that represent the function with the prescribed values at the grid points. For $j>0$, the variable $\alpha^{(j)}_{\boldsymbol{\ell},\mathbf{i}}$ is what we call "hierarchized up to dimension j", also referred to as the coefficient at position $(\boldsymbol{\ell},\mathbf{i})$ being in state j , and it is computed from the variables $\alpha^{(d-1)}_{*}$ by applying the 3-point stencil in direction j. More precisely, for a level index vector $(\boldsymbol{\ell},\mathbf{i})$ define the left hierarchical predecessor in direction r as $\mathscr{L}_r(\boldsymbol{\ell},\mathbf{i}) := (\boldsymbol{\ell}',\mathbf{i}')$, with $(\boldsymbol{\ell}'_r,\mathbf{i}'_r) := \mathscr{L}(\boldsymbol{\ell}_r,\mathbf{i}_r)$ and $(\boldsymbol{\ell}'_s,\mathbf{i}'_s) = (\boldsymbol{\ell}_s,\mathbf{i}_s)$ for $s\neq r$. $\mathscr{R}_r$ is defined analogously for the right hierarchical predecessor in direction r. With this we define $\alpha^{(j)}_{\boldsymbol{\ell},\mathbf{i}} = \alpha^{(j-1)}_{\boldsymbol{\ell},\mathbf{i}} - \frac{1}{2}\left(\alpha^{(j-1)}_{\mathscr{L}_j(\boldsymbol{\ell},\mathbf{i})} + \alpha^{(j-1)}_{\mathscr{R}_j(\boldsymbol{\ell},\mathbf{i})}\right)$, and for the boundary points in direction r with $(k_r,i_r)=(0,0)$ or $(k_r,i_r)=(0,1)$ we have $\alpha^{(j)}_{\boldsymbol{\ell},\mathbf{i}} = \alpha^{(j-1)}_{\boldsymbol{\ell},\mathbf{i}}$. If boundary points are not part of the task, for the sake of uniformity, we consider the modification of the boundary points as applying a 3-point stencil, too, only that the non-existent hierarchical predecessor variables are considered being 0. For a set of grid points U and a direction r let $\mathscr{H}_r(U)$ denote the set of hierarchical predecessors in direction r.

It is well known that one-dimensional hierarchization can be performed in-place by performing the hierarchization from high level to low level. It follows immediately that also high-dimensional hierarchization can be performed in-place by using the unidirectional principle. This is expressed in Algorithm 1, the classical unidirectional hierarchization algorithm. This formulation of the algorithm uses the notion of a pole, i. e., the grid points that are an axis-aligned one-dimensional grid in dimension k. In our notation, a pole in direction r is expressed as $\mathbf{G}_{\boldsymbol{\ell},I}$ where the interval I is such that $I_r=[0,1]$ for the direction r and all other components of I are single (grid) coordinates. We also use $\pi_r(\mathbf{G}_{\boldsymbol{\ell}})$ for the projection of $\mathbf{G}_{\boldsymbol{\ell}}$ along dimension r, i.e., replacing the r-th coordinate by 0. Therefore, $\pi_r(\mathbf{G}_{\boldsymbol{\ell}})$ contains exactly one grid point of each pole in direction r and can be used to loop over all poles in this direction.

4 Divide and Conquer Hierarchization

This section first derives the basic version of the divide and conquer hierarchization algorithm (Algorithm 2), proves its correctness and then analyzes its complexity. Subsequently, this algorithm is used as basic building block to derive hybrid algorithms

Algorithm 1: The unidirectional hierarchization algorithm.

```
1 Function unidirHierarchize(G_ℓ)
2     for r ← 1 to d do // unidirectional loop over dimensions
3         forall the x_{k,i} ∈ π_r(G_ℓ) do // loop over all poles in dimension r
4             for level ← ℓ_r down to 1 do // update pole bottom up
5                 k_r = level
6                 forall the index ∈ {1,...,2^{k_r} − 1} and index odd do
7                     i_r = index
8                     x_{k,i} = x_{k,i} − 0.5 * (x_{L_r(k,i)} + x_{R_r(k,i)})
```

which trade a weaker tall cache assumption for an increase in cache misses. The section ends with a sketch of parallelization possibilities for the derived algorithms.

In the new algorithm, Algorithm 2, we perform hierarchization in-place as in the unidirectional algorithm, but we do not follow the unidirectional principle globally. Like in the unidirectional algorithm We still have one intermediate result per grid point at any time, but the dimension up to which a grid point is hierarchized depends on its location.

The new algorithm divides the grid spatially and works on d-dimensional intervals (generalized axis parallel rectangle). The constituting one-dimensional interval may consist of a single grid point or be the support of a basis function corresponding to a grid point. More precisely, we call the d-dimensional interval $I = I_1 \times \cdots \times I_d$ a valid grid interval for the grid $\mathbf{G}_{\boldsymbol{\ell}}$ with respect to level vector $\boldsymbol{\ell}$ if all intervals are of the form $I_r = I_{k_r,i_r}$ or a single grid point $I_r = \{x_{k_r,i_r}\} = [x_{k_r,i_r}, x_{k_r,i_r}]$, with $k_r \leq \ell_r$. Now the subgrid of $\mathbf{G}_{\boldsymbol{\ell}}$ corresponding to the interval I is defined as

$$\mathbf{G}_{\boldsymbol{\ell},I} = \mathbf{G}_{\boldsymbol{\ell}} \cap I$$

Such valid grid intervals have few hierarchical predecessors outside of the interval itself which will ensure that the algorithm is efficient: if I_r is a singleton of the form $I_r = \{x_{k,i}\}$, then $I'_r := \{x_{\mathscr{L}(k,i)}, x_{\mathscr{R}(k,i)}\}$. Otherwise ($I_r$ is an open interval) set $I'_r := \overline{I_r}$. Complete the definition of I' by setting $I'_s = I_s$ for $s \neq r$. With that, the notion of hierarchical predecessors is extended to valid grid intervals by defining $\mathscr{H}_r(\mathbf{G}_{\boldsymbol{\ell},I}) := I'$. Subgrids $\mathbf{G}_{\boldsymbol{\ell},I}$ can be defined for arbitrary intervals I and are not restricted to valid grid intervals. Furthermore, we write the shorthand $\mathbf{G}_{\boldsymbol{\ell},\mathscr{H}_r(I)} := \mathscr{H}_r(\mathbf{G}_{\boldsymbol{\ell},I})$. To describe the hierarchical predecessors that are part of the 3-point stencil, it is convenient to define the boundary of an interval as $\mathscr{B}_r(I) = \mathscr{H}_r(I) \setminus I$, i. e., the hierarchical predecessors of I which are outside of I. Observe that these boundary points in different directions are disjoint, i.e., $\mathscr{B}_r(I) \cap \mathscr{B}_s(I) = \emptyset$ if $r \neq s$.

The number of grid points $s_r(\mathbf{G}_{\boldsymbol{\ell},I})$ (size) in dimension r of grid $\mathbf{G}_{\boldsymbol{\ell},I}$, $s_r(\mathbf{G}_{\boldsymbol{\ell},I})$ is the number of different coordinates in dimension r that occur for grid points in $\mathbf{G}_{\boldsymbol{\ell},I}$. For a valid grid interval I of $\mathbf{G}_{\boldsymbol{\ell}}$ we have

Fig. 3 *Left:* Applying the `boundarySplit` to the interval $[0,1]$. *Right:* Applying the `interiorSplit` to an interior grid interval $]\mathbf{x}_{\mathbf{k},\mathbf{j}}, \mathbf{x}_{\mathbf{k}',\mathbf{j}'}[$.

$$s_r(\mathbf{G}_{\boldsymbol{\ell},I}) = \begin{cases} 2^{\ell_r - k_r + 1} - 1 & \text{if } I_r = I_{k_r,i_r} \text{ (with } i_r \text{ odd)}, \\ 2^{\ell_r} + 1 & \text{if } I_r = [0,1], \\ 1 & \text{if } I_r \text{ is a single grid point.} \end{cases}$$

Next, we define the $\texttt{chooseDim}\big(\mathbf{G}_{\boldsymbol{\ell},I}\big)$ function for a grid $\mathbf{G}_{\boldsymbol{\ell},I}$. It returns the dimension for which the grid $\mathbf{G}_{\boldsymbol{\ell},I}$ has the most grid points (ties can be broken arbitrarily, e. g., choose the smallest dimension).

$$\texttt{chooseDim}\big(\mathbf{G}_{\boldsymbol{\ell},I}\big) = \operatorname*{arg\,max}_{1 \le r \le d} \left\{ s_r(\mathbf{G}_{\boldsymbol{\ell},I}) \right\} .$$

To split the multidimensional interval $I = I_1 \times \cdots \times I_d$ in direction r into three parts we define the following functions. The split functions rely upon $\mathbf{G}_{\boldsymbol{\ell},I}$ containing more than one grid point in direction r, i.e. $k_r < \ell_r$. For all dimensions $s \neq r$ we set $I_s^* = I_s$ (for $* \in \{0, int, 1, \text{left}, \text{mid}, \text{right}\}$). The case where $I_r = [0,1]$ is the only situation where an outer boundary should be split off, hence we set $I_r^0 = \{0\}$, $I_r^{int} =]0,1[$ and $I_r^1 = \{1\}$, and write

$$\texttt{boundarySplit}\big(r, I\big) := \left(I_r^0, I_r^{int}, I_r^1\right) .$$

Otherwise $I_r = I_{k,i}$, and we set $I_r^{\text{left}} = I_{k+1,2i-1} =]x_{\mathscr{L}(k,i)}, x_{k,i}[$, $I_r^{\text{mid}} = \{x_{k,i}\}$ and $I_r^{\text{right}} = I_{\ell+1,2i-1} =]x_{k,i}, x_{\mathscr{R}(k,i)}[$. We write

$$\texttt{interiorSplit}\big(r, I\big) := \left(I_r^{\text{left}}, I_r^{\text{mid}}, I_r^{\text{right}}\right) .$$

Both splits are depicted in Figure 3. Clearly, the three parts are a partitioning of the subgrid, and because only non-trivial dimensions are split, they are all non-empty.

With these definitions, Algorithm 2 is well defined. Its call structure is illustrated in Figure 4. Next we show that a call $\texttt{hierarchizeRec}\big(0, d, \mathbf{G}_{\boldsymbol{\ell}}, [0,1]^d\big)$ hierarchizes the grid correctly. Subsequently, the complexity of the algorithm is analyzed.

4.1 Correctness

To prove the correctness of Algorithm 2 it is sufficient to show that whenever we apply the 3-point stencil in direction r, all three participants store the value $\alpha^{(r-1)}$ and that, in the end, the 3-point stencils in all d dimensions have been applied for

Algorithm 2: Divide and conquer hierarchization algorithm.

```
1  Function hierarchizeRec(s,t,G_ℓ,I)
       // I is a valid grid interval for G_ℓ. Initially, all
          variables
       // of G_{ℓ,I} store the values α^(s), in the end they store α^(t).
       // This function changes only variables of G_{ℓ,I}.
       // It assumes that G_{ℓ,B_r(I)}, i.e., the boundary points of I
          in
       // direction r, stores the values α^(r).
2      if G_{ℓ,I} = {x_{k,i}} then // grid consists of a single grid point
3          for r ← (s+1) to t do
4              x_{k,i} = x_{k,i} − 0.5 * (x_{l_r(k,i)} + x_{r_r(k,i)})
5      else // i.e., |G_{ℓ,I}| > 1
6          r = chooseDim(G_{ℓ,I})
           // split G into subgrids in dimension r
7          if I_r = [0,1] then // case of global boundary
8              (I^0, I^int, I^1) = boundarySplit(r,I)
9              hierarchizeRec(s,(r−1),G_ℓ,I^0)
10             hierarchizeRec(s,(r−1),G_ℓ,I^1)
11             hierarchizeRec(s,t,G_ℓ,I^int)
12             hierarchizeRec((r−1),t,G_ℓ,I^0)
13             hierarchizeRec((r−1),t,G_ℓ,I^1)
14         else // I_r ⊂ (0,1), i.e., no global boundary
15             (I^left, I^mid, I^right) = interiorSplit(r,I)
16             hierarchizeRec(s,(r−1),G_ℓ,I^mid)
17             hierarchizeRec(s,t,G_ℓ,I^left)
18             hierarchizeRec(s,t,G_ℓ,I^right)
19             hierarchizeRec((r−1),t,G_ℓ,I^mid)
```

all grid points. The presented argument does not rely on the regular structure of the component grids and hence works for regular and adaptively refined sparse grids identically.

When an interval $I_{\mathbf{k},\mathbf{i}}$ is split, the boundary of the resulting subintervals is either part of the boundary of $I_{\mathbf{k},\mathbf{i}}$ or lies exactly in the middle of $I_{\mathbf{k},\mathbf{i}}$:

Lemma 2. *Let $I = I_{\mathbf{k},\mathbf{i}}$ for $\mathbf{k} \geq 1$ be a valid interval for some grid and assume I is split by $\left(I^{left}, I^{mid}, I^{right}\right) := \texttt{interiorSplit}\left(r, I\right)$.*

1. *In directions different from r, the boundary remains:*
   If $s \neq r$ we have $\mathscr{B}_s(I^{left}) \subset \mathscr{B}_s(I)$, $\mathscr{B}_s(I^{right}) \subset \mathscr{B}_s(I)$, and $\mathscr{B}_s(I^{mid}) \subset \mathscr{B}_s(I)$.
2. *The boundary of full-dimensional parts is the boundary or the split-plane:*
   $\mathscr{B}_r(I^{left}) \subset \left(\mathscr{B}_r(I) \cup I^{mid}\right)$, and $\mathscr{B}_r(I^{right}) \subset \left(\mathscr{B}_r(I) \cup I^{mid}\right)$.
3. *The hierarchical predecessors of the split plane is the old boundary:*
   $\mathscr{B}_r(I^{mid}) = \mathscr{B}_r(I)$.

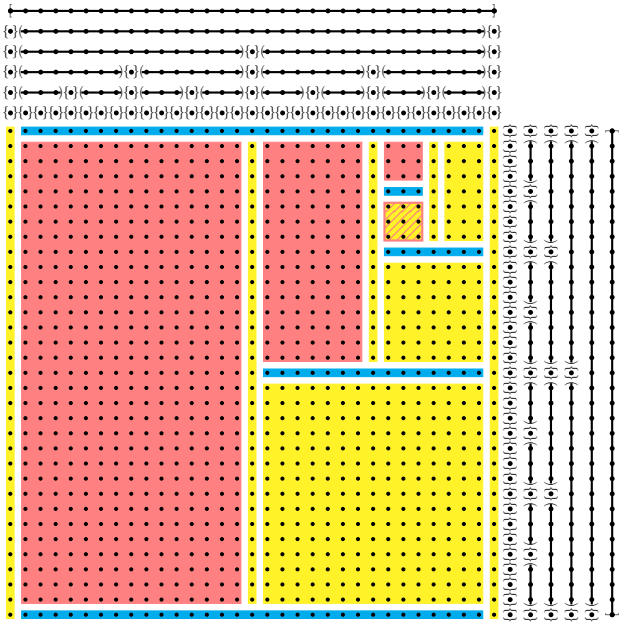

Fig. 4 Hierarchizing a 2-dimensional grid with the divide and conquer hierarchization algorithm (Algorithm 2). The yellow (or blue or red) grid points are hierarchized up to dimension 0 (or 1 or 2), i. e., store the original functional values $\alpha^{(0)}_{\mathbf{k},\mathbf{i}}$ (or the values $\alpha^{(1)}_{\mathbf{k},\mathbf{i}}$ or the hierarchical surpluses $\alpha^{(2)}_{\mathbf{k},\mathbf{i}}$, respectively). The subgrid that is currently updated is hatched (in red). The algorithm progresses by hierarchizing complete grid intervals (depicted on the sides).

Proof. Follows from the definitions and Lemma 1 □

For the correctness of Algorithm 2 observe that the purpose of the call is to hierarchize the points inside I in direction $s+1,\ldots,t$. Accordingly, all points inside I are assumed to be already hierarchized in directions up to s, i.e., the variables present $\alpha^{(s)}$. We show that the following invariants hold for the recursion.

For a call of `hierarchizeRec(`$s,t,\mathbf{G}_\ell,I$`)` we formulate the following states of the grid $\mathbf{G}_\ell$.

Definition 1 (Precondition($\mathbf{G}_\ell, I, s$)).

1. a variable in $\mathbf{G}_{\ell,I}$ holds the value $\alpha^{(s)}$.
2. a variable in $\mathbf{G}_\ell \cap \mathscr{B}_r(I)$ holds the value $\alpha^{(r)}$.

Definition 2 (Postcondition($\mathbf{G}_\ell, \mathbf{G}'_\ell, I, t$)).

1. a variable in $\mathbf{G}'_{\ell,I}$ holds the value $\alpha^{(t)}$.
2. all other variables have the same value in $\mathbf{G}'_\ell$ as in $\mathbf{G}_\ell$.

Lemma 3. *If* `hierarchizeRec(`$s,t,\mathbf{G}_\ell,I$`)` *is called in a situation as described by Definition 1, then the grid* $\mathbf{G}'_\ell$ *after the call is as described by Definition 2.*

Proof. By induction on size of the current subgrid, i.e., the number of grid points in $\mathbf{G}_{\boldsymbol{\ell},I}$. First, consider the case that the current subgrid $\mathbf{G}_{\boldsymbol{\ell},I}$ contains a single grid point. Then the for-loop in Line 3 changes the state from the first item of the precondition to the first item of the postcondition. In this, the stencil in Line 4 is correct because of the second item of the precondition.

Second, if $\left|\mathbf{G}_{\boldsymbol{\ell},I}\right| > 1$, then the algorithm performs a split operation. By Lemma 2, Item 1, the invariant on the boundary in directions different from r is transferred from the call to all recursive calls. Because the split partitions $\mathbf{G}_{\boldsymbol{\ell},I}$ into three subgrids, Item 1 of the precondition is also transferred. The algorithm distinguishes between 2 further cases with $\left|\mathbf{G}_{\boldsymbol{\ell},I}\right| > 1$, namely $I_r = [0,1]$ or $I_r \subset (0,1)$.

In the first case we have $I_r = [0,1]$. In this case the boundary of I^0 and I^1 in direction r is empty, i.e., $\mathscr{B}_r(I^0) = \mathscr{B}_r(I^1) = \emptyset$. Hence, Item 2 of the precondition holds trivially for the calls in Lines 9, 10 and 12, 13. As $\mathscr{B}_r(I^{int}) = \left(I^0 \cup I^1\right)$, it follows from the postcondition of the first two calls that Item 2 of the precondition for the call in Line 11 is fulfilled. Item 1 of the precondition for the call in Line 12 follows from Item 1 of the postcondition of the call in Line 9. A similar argument works for Line 13.

Otherwise I_r is an open interval that contains at least 3 grid points of $\mathbf{G}_{\boldsymbol{\ell}}$. Hence, the split operation partitions the subgrid into three non-empty subgrids. Item 2 of the precondition for the call in Line 16 follows directly from Lemma 2, Item 3. This establishes by Lemma 2, Item 2 the precondition for the calls in Line 17 and Line 18. Item 1 of the precondition for the call in Line 19 follows from Item 1 of the postcondition of the call in Line 16.

The postcondition on the final grid $\mathbf{G}'_{\boldsymbol{\ell}}$ for the whole call now follows from the postconditions of the recursive calls. □

Therefore, the call `hierarchizeRec`$\left(0,d,\mathbf{G}_{\boldsymbol{\ell}},[0,1]^d\right)$ hierarchizes the whole grid $\mathbf{G}_{\boldsymbol{\ell}}$ correctly as the precondition holds trivially and the postcondition means that all grid points are correctly hierarchized.

4.2 Complexity Analysis

For each grid point Algorithm 2 performs exactly d updates and for each update precisely 3 grid points are needed. As at most 3 cache lines are needed in cache simultaneously, the hierarchization of any grid point does not cause more than $3d$ cache misses, i. e., 3 cache misses per update, for any reasonable cache replacement strategy. If the subgrid the algorithm works on is sufficiently small, i. e., the subgrid and all hierarchical predecessors of the stencil fit it into memory, the algorithm is much more efficient. In that case the whole subgrid can be hierarchized by loading it and its hierarchical predecessors into memory once and performing the updates in memory. As the subgrid fits into cache and no other cache lines are accessed in between, a LRU strategy ensures that the subgrid stays in cache as long as it is needed. The analysis builds upon these observations.

Note that for all d-dimensional grids, i. e., for all grids that have more than 1 grid point in each dimension, the call $\texttt{hierarchizeRec}\big(s,t,\mathbf{G}_{\boldsymbol{\ell},I}\big)$ always happens with $s=1$ and $t=d$. The parameters s and t are only altered for the grids of the form $\mathbf{G}_{\boldsymbol{\ell},I_r^0}$, $\mathbf{G}_{\boldsymbol{\ell},I_r^1}$ and $\mathbf{G}_{\boldsymbol{\ell},I_r^{mid}}$ which have at least one dimension with a single grid point.

The presented analysis assumes that the grid $\mathbf{G}_{\boldsymbol{\ell}}$ is significantly larger than the cache. In particular, it is assumed that all directions are refined such that there exist isotropic subgrids, i. e., subgrids with the same number of grid points in each dimension, that do not fit into cache. For component grids, this is the case if $\left(2^{\min_r \ell_r}-1\right)^d \geq M$, where M is the memory size.

We analyze the performance of the algorithm by focusing on certain calls to $\texttt{hierarchizeRec}\big(0,d,\mathbf{G}_{\boldsymbol{\ell}},I_i\big)$ on the same level of the recursion, given by the family of intervals $I_i, i\in F$, where $F=\{(\boldsymbol{\ell},\mathbf{i}) \mid \boldsymbol{\ell}=(m,\ldots,m)\}$. All I_i have the same shape and size. We choose them in a way that the level of the subgrid in each dimension is m. With our particular definition of $\texttt{chooseDim}\big(\mathbf{G}_{\boldsymbol{\ell},I}\big)$ and the sufficiently large grids we consider, these calls are actually performed. The intervals intervals I_i $(i\in F)$ almost partition the domain, namely they are disjoint and the union of their closures is the complete domain, i.e., $\cup_i \overline{I_i}=[0,1]^d$. Note that $\mathbf{G}_{\boldsymbol{\ell},\overline{I_i}\setminus I_i}$ contains the boundary points in all directions and a few more points.

The analysis of such a call is based on the number of grid points of the subgrid in its interior $N(m)=|\mathbf{G}_{\boldsymbol{\ell},I_i}|$ and on its boundary $Q(m)=|\mathbf{G}_{\boldsymbol{\ell},\overline{I_i}\setminus I_i}|$. More precisely, we need a good lower bound on $N(m)$ (progress), and good upper bounds on $N(m)+Q(m)$ (base cost and memory requirement) and $Q(m)$ (additional cost). For the base cost and memory requirement we additionally, have to take into consideration the layout of the grid and how it interacts with the blocks of the (external) memory.

Once we identified an m such that the whole grid $\mathbf{G}_{\boldsymbol{\ell},\overline{I_i}}$ fits into memory, we can estimate the overall number of cache misses in the following way: the call $\texttt{hierarchizeRec}\,(0,d,\mathbf{G}_{\boldsymbol{\ell}},I_i)$ hierarchizes $\mathbf{G}_{\boldsymbol{\ell},I_i}$ and costs loading the subgrid and its boundary. The number of cache misses is $(N(m)+Q(m))/B$ plus an additional term that is less then $Q(m)$ to account for blocks that are not completely filled (assuming a row major layout). Hierarchizing the boundary $\mathbf{G}_{\boldsymbol{\ell},\overline{I_i}\setminus I_i}$ incurs at most $3d$ cache misses per boundary point. The number of calls can be estimated by $|F|<|\mathbf{G}_{\boldsymbol{\ell}}|/N(m)$. Hence, the total number of cache misses is at most

$$\frac{|\mathbf{G}_{\boldsymbol{\ell}}|}{N(m)}\left(\frac{N(m)+Q(m)}{B}+(3d+1)Q(m)\right)\leq|\mathbf{G}_{\boldsymbol{\ell}}|\left(\frac{1}{B}+\frac{(3d+2)Q(m)}{N(m)})\right)$$

Hence, in the following we analyze the asymptotic behavior of the additional term $\frac{(3d+2)Q(m)}{N(m)}$, and show that it is $o(1/B)$.

Lemma 4. *Hierarchizing a component grid $\mathbf{G}_{\boldsymbol{\ell}}$ with $\ell_r\geq\frac{1}{d}\log_2 M$ $(\forall r)$ using Algorithm 2 takes*

$$|\mathbf{G}_{\boldsymbol{\ell}}|\left(\frac{1}{B}+\mathcal{O}\left(\frac{1}{\sqrt[d]{M}}\right)\right)$$

cache misses in the cache-oblivious model with the tall cache assumption $B=o(\sqrt[d]{M})$ and an LRU cache replacement strategy.

Proof. In the setting of component grids we have $N(m) = (2^m - 1)^d$, and $Q(m) \leq 2d(2^m+1)^{d-1}$. We choose

$$m = \log_2 \left(\sqrt[d]{M/2} - 1 \right) .$$

leading to

$$\frac{1}{N(m)} = \frac{1}{(2^m-1)^d} = O\left(\frac{1}{M}\right) \text{ and } Q(m) \leq 2d \left(\sqrt[d]{M/2} \right)^{d-1} = O\left(M^{\frac{d-1}{d}}\right) .$$

From the tall cache assumption we conclude that for any constant c we have $c \cdot B^d \leq M$ for large enough M and B, which we use as $B \leq \frac{1}{2} \cdot \sqrt[d]{M/2}$. Assuming a row major layout, the number of occupied cache lines is upper bounded by

$$\begin{aligned}
&\left(\left\lceil \frac{2^m+1}{B} \right\rceil + 1 \right) \cdot (2^m+1)^{d-1} \leq \frac{(2^m+1)^d}{B} + 2 \cdot (2^m+1)^{d-1} = \\
&= \frac{M}{2 \cdot B} + 2^{1/d} \cdot M^{\frac{d-1}{d}} \leq \frac{M}{2 \cdot B} + 2^{1/d} \cdot M^{\frac{d-1}{d}} \cdot \underbrace{\frac{M^{1/d}}{2^{\frac{d+1}{d}} B}}_{\geq 1} = \frac{M}{B} .
\end{aligned}$$

Hence, the choice of m is as required in the preceding discussion. In that case the additional term is

$$\frac{(3d+2)Q(m)}{N(m)} = O\left(\frac{M^{\frac{d-1}{d}}}{M} \right) = O\left(\frac{1}{\sqrt[d]{M}} \right) = o\left(\frac{1}{B} \right) ,$$

where the last equality is the tall cache assumption. □

When the constant d is made explicit in the last equation the lower order term reads $\mathcal{O}\left(d^2 / \sqrt[d]{M}\right)$.

4.3 Hybrid Algorithms

One aspect of Algorithm 2 and its analysis that might limit its applicability is the fairly strong tall cache assumption $B = o\left(\sqrt[d]{M}\right)$. Algorithm 1, in contrast, can be modified to work with $|\mathbf{G}_\ell| \left(\frac{d}{B} + \mathcal{O}\left(\frac{1}{M}\right) \right)$ cache misses if $B = o(M)$. In fact, these two algorithms mark the corners of a whole spectrum of algorithms that become more cache efficient as the cache gets taller. Instead of working subsequently in all d directions as Algorithm 1, or merging all d phases as Algorithm 2, these hybrid algorithms merge $c \in \mathbb{N}$, $1 \leq c \leq d$ phases of the unidirectional principle. To discuss these hybrid algorithms it is first assumed that the component grid $\mathbf{G}_\ell$ is stored in a block aligned fashion, i. e., every pole in direction 1 is padded with dummy elements such that every pole starts at the beginning of a cache line. As a result, all poles are

split into cache lines in the very same way. After discussing this aligned case, the hybrid algorithms are also sketched for the case that the alignment is not possible.

Lemma 5. *For every $c \in \mathbb{N}$, $1 \leq c \leq d$ and c divides d there is an algorithm that, assuming a tall cache with $M = \omega(B^c)$, performs hierarchization on a block aligned component grid $\mathbf{G}_\ell$ with $|\mathbf{G}_\ell| \left(\frac{d}{c} \cdot \frac{1}{B} + \mathcal{O}\left(\frac{1}{\sqrt[c]{M}}\right)\right)$ cache misses.*

Proof. Let us first consider the case of $c = 1$, i.e., the mentioned modification of Algorithm 1. To hierarchize a pole, replace Line 4 to Line 8 by the call `hierarchizeRec(`$r-1, r, \mathbf{G}_\ell, I$`)` for the one-dimensional poles e.g. $I = [0,1] \times \{x_{k_2,i_2}\} \times \cdots \times \{x_{k_d,i_d}\}$.

For $r = 1$ and the considered row major layout, the poles are contiguous in memory. Therefore, each pole can be considered as a 1-dimensional subgrid to which Algorithm 2 is applied. Therefore, Lemma 4 yields that this modification of Algorithm 1 needs $|\mathbf{G}_\ell| \left(\frac{1}{B} + \mathcal{O}\left(\frac{1}{M}\right)\right)$ cache misses for the first unidirectional pass, i. e., to hierarchize the first dimension. In that case, the tall cache assumption of Lemma 4 is $B = o(M)$.

For $r > 1$ the poles worked on are not stored contiguously in memory, and working on a single pole at a time would access a whole cache line to only work with a single element. This can be avoided by the following kind of blocking that works with the poles in direction r that share the same cache line. These poles are by layout neighboring in direction 1. For the sake of the formulation of the algorithm and its complexity analysis, this allows us to consider the cache lines instead of the grid points as the atomic elements, which results in cache line size $B' = 1$, internal memory size $M' = M/B$ and grid size $\mathbf{G}'_\ell = \mathbf{G}_\ell / B$ (all in cache lines). As we now consider $B' = 1$, the memory can be filled with the new (meta-)poles without polluting the internal memory with other grid points. Therefore the analysis is identical to the case of $r = 1$ which yields that transferring the grid from state $\alpha^{(r-1)}$ to state $\alpha^{(r)}$ needs $|\mathbf{G}_\ell| \left(\frac{1}{B} + \mathcal{O}\left(\frac{1}{M}\right)\right)$ cache misses and a tall cache assumption of $B = o(M)$.

For $c > 1$ the one-dimensional poles are replaced by c-dimensional planes. The hybrid algorithm works in d/c phases and each phases hierarchizes the grid from state $\alpha^{((p-1)\cdot c)}$ to state $\alpha^{(p \cdot c)}$ for some $p \in \mathbb{N}$. If $c = 2$, the interval is for example $I = [0,1] \times [0,1] \times \{x_{k_3,i_3}\} \times \cdots \times \{x_{k_d,i_d}\}$ and this I can be regarded as a 2-dimensional pole which can be hierarchized in dimension 1 and 2 by the call `hierarchizeRec(`$0, 2, \mathbf{G}_\ell, I$`)`. The hybrid algorithm performs this call for all such intervals, bringing the complete grid to the state $\alpha^{(c)}$.

For $p = 1$, the intervals are in contiguous memory and the c-dimensional analysis of Lemma 4 applies. This shows that transforming the grid from state $\alpha^{(0)}$ to state $\alpha^{(c)}$ takes $|\mathbf{G}_\ell| \left(\frac{1}{B} + \mathcal{O}\left(\frac{1}{\sqrt[c]{M}}\right)\right)$ cache misses and requires the tall cache assumption $B = o(\sqrt[c]{M})$.

For $p > 1$, i.e., if the intervals are orthogonal to the direction of the layout, we can again use a version of the algorithm that works on B intervals simultaneously, i. e., regards the cache lines as the atomic elements instead of the grid points (i. e., cache line size $B' = 1$, internal memory size $M' = M/B$ and grid size $\mathbf{G}'_\ell = \mathbf{G}_\ell / B$ (all in cache lines)). As we consider the case $B' = 1$, the same analysis as in the

case of hierarchizing the first c dimensions, i. e., $p = 1$, can be applied. This yields that transforming the grid from state $\alpha^{((p-1)\cdot c)}$ to state $\alpha^{(p\cdot c)}$ for any $p > 1$ causes $\frac{|\mathbf{G}_\ell|}{B}\left(\frac{1}{1} + \mathcal{O}\left(\sqrt[c]{\frac{B}{M}}\right)\right) = |\mathbf{G}_\ell|\left(\frac{1}{B} + \mathcal{O}\left(\frac{1}{\sqrt[c]{M \cdot B^{c-1}}}\right)\right)$ cache misses. The tall cache assumption required to use the analysis of the $p = 1$ case is $B = o(M)$. This assumption also guarantees that the second term is of lower order.

As this hybrid algorithm performs d/c sweeps over the complete grid and the lower order term for $p = 1$ dominates that of $p > 1$, i. e., $\frac{1}{\sqrt[c]{M \cdot B^{c-1}}} = \mathcal{O}\left(\frac{1}{\sqrt[c]{M}}\right)$ given $B = o(M)$, the statement of the lemma follows. □

Lemma 6. *For $1 \leq u < d$ and assuming a tall cache of $M = \omega(B^u)$, the number of cache misses to hierarchize a block aligned component grid $\mathbf{G}_\ell$ is*

$$|\mathbf{G}_\ell|\left(\frac{2}{B} + \mathcal{O}\left(\frac{1}{\sqrt[u]{M}}\right) + \mathcal{O}\left(\frac{1}{\sqrt[d-u]{M \cdot B^{d-u-1}}}\right)\right).$$

Proof. Consider a hybrid algorithm which works in 2 passes, each pass hierarchizing a different number of dimensions: the first pass hierarchizes the first u dimensions, i. e., $c = u$ and $p = 1$. The second pass hierarchizes the last $(d-u)$ dimensions, i. e., $c = (d-u)$ and $p > 1$. It follows from the proof of Lemma 5 that the first pass requires a tall cache assumption of $B = o\left(\sqrt[u]{M}\right)$ and causes $|\mathbf{G}_\ell|\left(\frac{1}{B} + \mathcal{O}\left(\frac{1}{\sqrt[u]{M}}\right)\right)$ cache misses. Also by the proof of Lemma 5, the second pass requires a tall cache assumption of $B = o(M)$ and causes $|\mathbf{G}_\ell|\left(\frac{1}{B} + \mathcal{O}\left(\frac{1}{\sqrt[d-u]{M \cdot B^{d-u-1}}}\right)\right)$ cache misses. □

For $u \geq d/2$, it holds that $\frac{1}{\sqrt[d-u]{M \cdot B^{d-u-1}}} = \mathcal{O}\left(\frac{1}{\sqrt[u]{M}}\right)$ such that the first lower order term in Lemma 6 dominates. As the tall cache assumption $M = \omega(B^u)$ just becomes stronger as u increases, choosing $u > d/2$ is therefore not advantageous. For $u < d/2$, it depends on the actual size of the cache whether the first or the second lower order term dominates. In particular, for $u = 1$, Lemma 6 becomes:

Lemma 7. *Assuming a cache of size $M = \omega(B)$, a block aligned component grid $\mathbf{G}_\ell$ can be hierarchized with $|\mathbf{G}_\ell|\left(\frac{2}{B} + \mathcal{O}\left(\frac{1}{\sqrt[d-1]{M \cdot B^{d-2}}}\right)\right)$ cache misses.*

If for some reason a block aligned layout of the component grid is not feasible, the hybrid algorithms can block b poles together. When b is sufficiently larger than B (i. e., $B = o(b)$), then there are at most two cache lines which contain also unused grid points for every $\lfloor b/B \rfloor - 1$ full cache lines. This changes the term $1/B$ to $(1/B + 3/b)$ in the above analysis, adding another lower order term, and the effective size of the cache to $M' = M/b$.

4.4 Parallelization

To achieve high performance on modern machines, it is important that an algorithm can use many parallel processors. In Algorithm 2 this is possible by executing

the two recursive calls in Line 17 and Line 18 in parallel. This is still a correct algorithm because I_r^{left} and I_r^{right} are disjoint and the precondition for both calls is already established after Line 16. On the level of grid points, i. e. $B = 1$, the resulting algorithm implements an "exclusive write" police, i.e., two different processors never write to the same memory location simultaneously. Without further synchronization it requires the possibility of "concurrent read" because both parallel calls read the variables in $\mathbf{G}_{\boldsymbol{\ell}, I_r^{mid}}$.

Considering cache-lines, it is possible that two different processors write to the same cache line. To avoid this, the algorithm performs the two calls serially if and only if the split was done in dimension 1, the direction in which a cache line extends. Hierarchization of the boundaries only needs $O(|\mathbf{G}_{\boldsymbol{\ell}}|/M)$ cache misses, even if it would be performed in serial. On a system with P processors, each having a private cache of size M, the above version of Algorithm 2 achieves that the number of parallel cache misses is $|\mathbf{G}_{\boldsymbol{\ell}}| \left(\frac{1}{PB} + \mathcal{O}\left(\frac{1}{\sqrt[d]{M}} \right) \right)$ as long as $P \leq \prod_{r=2}^{d} 2^{\boldsymbol{\ell}_r}$.

5 Experimental Evaluation

In this section we report on the run times of our implementation of the described recursive algorithm, its variants and alternatives. The experiments confirm that the main bottleneck of the task is the memory access. We conclude this from the observation that the measured running times are generally close to what the I/O model predicts. Hence, further improvement can only be expected when implementing a different I/O algorithm. Still, it should be noted that this is only true once the implementation is sufficiently carefully optimized in other aspects, most notably multicore parallelism and vectorization, but also branch mispredictions, overhead for (recursive) function calls, and the creation of parallel tasks. We also observe that with increasing dimension the gap between the prediction and the measurements increases, which we suppose has several reasons: Our analysis is not particularly careful with respect to higher dimensions and constant memory size. The I/O-model ignores additional memory effects like the TLB, i.e., the cache used to perform virtual memory translation.

5.1 Setup and systems

We implemented the algorithm in C++, using openMP for parallelization and hand coded AVX-vectorization. In the implementation we use the C-style numbering of the dimensions starting with 0, but in the description here we translate this to the usual numbering from 1 to d. The experiments are performed for the case without boundary points, i.e., where all global boundary points are implicitly 0.0. For the recursive algorithms, this is actually more complicated than if all boundaries are

present because the recursive calls create some cases with boundaries and hence it is necessary to keep track of the existence of boundaries in the different directions. This allows a direct comparison with [24].

The main focus of the experiments is wall-clock run-time as measured by the chrono timer provided in C++11. This is usually taken relative to the time it takes to touch the grid once, as calculated from the measured performance by the stream benchmark (using as many cores as helpful), multiplied with the size of the grid. The stream benchmark measures the speed at which the CPU can access large amounts of data that is stored contiguously. It turned out to be sensitive to the used compiler, so we always took the highest performance reported.

The experiments do not empty the cache in order to provide cold cache measurements, but by the size of the grid and the structure of the algorithm, the influence of the content of the cache when the measurement starts is small.

The implementation is compiled with gcc in the version available on the architecture (see below), using the flags `-mavx -Wa,-q -Wall -fopenmp -std=c++11 -march=native -O3`. We use openMP to run the code on several cores in parallel, using the static scheduling to distribute the work of for loops, and the concept of tasks for the recursive algorithm. Our code compiles with icc, but a small set of test runs showed that for the bigger grids we are interested in, there were hardly any differences to gcc.

5.1.1 System 1: Rechenteufel

Most of the experiments were performed on this system. It is a standalone workstation (called Rechenteufel) with an IvyBridge Intel(R) Xeon(R) CPU E3-1240 V2 3.40GHz. with 8 MB shared L3 cache. (From IvyBridge Specs: private L2 Cache of 256 KB, private L1 Cache of 64 KB.) It has 1 CPU with 4 cores (no hyperthreading) and 32GB DDR3 main memory. The stream benchmark using icc version 13.1.3 (gcc version 4.7.0 compatibility) gives a performance of 21.9 GB/s = $21.9 \cdot 10^9$ byte/s. The used gcc has version 4.8.3. For the reported experiments the maximum grid sizes are roughly 8 GB (a quarter of the main memory).

5.1.2 System 2: Hornet

This system is used for the experiments with GENE. It is one node of a supercomputer called hornet. The CPU is an Intel Haswell E5-2680v3 2,5 GHz with 12 Cores, hyperthreading off, and 30 MB shared L3 cache. (from haswell sepcs: L2 cache: 256 KB per core, L1 cache 64 KB per core), and it has 64 GB DDR4 main memory. One node has two such CPUs, but our experiments only used one of them.

The stream Benchmark with cc (cray compiler) version 8.3.6, using 12 cores, gives $57.286 \cdot 10^9$ Bytes/s.

5.2 Compared Algorithms

Our experimental evaluation considers the task of hierarchization without boundary points.

5.2.1 Unidirectional Algorithm

The basic unidirectional algorithm has been implemented very efficiently as described in [24]. It has a natural lower bound of d times scanning, which is almost achieved in many cases.

5.2.2 Recursive Algorithm

This is an implementation of Algorithm 2, as explained and analyzed in Section 4. Hence, for sufficiently big cache (compared to the dimension), this algorithm scans the data set once.

To reduce the overhead of recursive execution, we use as base case regions (sub-)poles in dimension 1 of level `recTile`. In our data layout, such a region is consecutive in memory. Hierarchization in dimension 1 needs two additional values and is done iteratively and without vectorization. In contrast, each hierarchization in a dimension different from 1 needs two other such regions, and the application of the stencil is done vectorized using AVX instructions on 4 doubles.

Multi-core parallelism is implemented as described in Section 4 using openMP tasks. To avoid the task creation overhead for very small tasks, we do not create tasks if the level sum of the current recursive call is too small. With a focus on the shared level 3 cache, we also do not parallelize the tasks if the level sum is too big. These limits are called `minSpawnLevel` and `maxSpawnLevel`.

Given that this algorithm has the three parameters `recTile`, `minSpawnLevel` and `maxSpawnLevel`, we conducted a parameter study that lead to a reasonable heuristic to choose these parameters. This algorithm turned out to be the fastest for problems with 2,3 or 4 dimensions.

5.2.3 Hybrid Algorithm: Twice Rec

This is an implementation of the hybrid Algorithm described in Section 4.3 that performs two scans over the data set. The first phase considers meta-poles in dimensions 1 to d_{split}, i.e. $1 \leq d_{\text{split}} < d$. These are small complete d_{split} dimensional component grids, presumably small enough to fit into cache. They are hierarchized iteratively using the optimized unidirectional algorithm of [24]. The loop over these subgrids is parallelized.

The second phase is Algorithm 2 operating on vectors that constitute the hierarchized d_{split} dimensional subgrids. Here, the base case is a single vector, and the

application of the stencil is vectorized. To increase the number of vectors that can fit into cache, we split the vectors into chunks of `BlockSize` elements. But, to amortize the overhead of the recursive call structure to sufficiently big base cases, we should not choose `BlockSize` too small. Hence for the second phase we have the parameters `BlockSize`, `minSpawnLevel` and `maxSpawnLevel`, and for the overall algorithm the additional d_{split}. Again, a parameter study lead to a heuristic to choose good values for these parameters, namely to use $\texttt{BlockSize} = 1024$ and choose d_{split} in a way that the level sum of the subgrids is at least 14.

This algorithm currently gives the best performance for dimensions 5 and 6.

5.3 Parameter Study and Heuristics

The study of the parameters `minSpawnLevel`, `maxSpawnLevel`, and `recTile` for the recursive and `BlockSize` and d_{split} for the hybrid Algorithm needs to consider a big parameter space. Hence, we did not include further parameters and always used 8 openMP threads and grids of size 8GB. Further, we did not repeat the individual runs. Accordingly, the heuristic to choose the parameters might not be perfect, but as we will see in the next sections, the performance achieved with this heuristic is usually already pretty good.

5.3.1 Recursive

The parameters of the recursive algorithms are all depending on the level sum of the current rectangle. Hence, they can be at most $\texttt{maxLevel} = \sum_{i=2}^{d} \boldsymbol{\ell}_i$. In the parameter study we vary `maxSpawnLevel` between 4 and `maxLevel`, and `minSpawnLevel` between 3 and $\texttt{maxSpawnLevel} - 1$. The parameter `recTile` varies between 2 and $\boldsymbol{\ell}_1$.

The following heuristic yields performance that is close to the best choice of parameters: We chose `recTile` to be $\boldsymbol{\ell}_1$ if this is smaller than 14, else we choose it to be 5. Further we choose $\texttt{maxSpawnLevel} = \texttt{maxLevel} - 2$, $\texttt{minSpawnLevel} = \texttt{maxLevel} - 7$. This choice of parameters is reasonable in the following sense: A real split in the first dimension is expensive as on the boundary we access a whole cache line to use a single grid point. Therefore, if the first dimension is small, it is better to not split it at all. If the first dimension is very large, then the `recTile` is chosen rather small such that the tiles are more quadratic and the interior to boundary ratio is better than for tiles with a very long first dimension. Only recursive calls between `maxSpawnLevel` and `minSpawnLevel` are parallelized. Hence, this difference needs to be at least the binary log of the intended number of parallel tasks, explaining the difference of 5. Keeping `maxSpawnLevel` slightly away from `maxLevel` leads to all threads working on some (still big) subgrid, which seems beneficial, perhaps because of caching effects in the virtual address

translation (TLB). Generally we observe that the performance was not very sensitive to the choice of the spawn levels.

5.3.2 Twice Recursive

The algorithm is only meaningful if the parameter split dimension is in the interval $1 \leq d_{\text{split}} < d$, and the parameter study explores this whole range. The block size `BlockSize` is set to all powers of 2 between 4 and $\left(\prod_{i=1}^{d_{\text{split}}} 2^{\ell_i}\right) - 2$.

This lead to the heuristic of choosing d_{split} as the smallest dimension i such that $l = \sum_{j=1}^{i} \ell_j > 13$, and choosing `BlockSize` $= 1024$. With this heuristic, the subgrids of the first phase are at least $2^{14+3} bytes = 128$KB big (and not too much bigger). Hence, they can fit into the private L2 cache of 256KB if $l \leq 15$, and will fit into the 8 MB big L3 cache even if all four cores are active if $l \leq 18$. The choice of `BlockSize` is a reasonable compromise between keeping the memory requirement small and having enough work to amortize the overhead of the recursion.

5.3.3 Data and Results for Parameter Study

The results of the parameter study for the recursive algorithm are shown in Table 1, that of the twice recursive algorithm in Table 2. Comparing the running times between the tables shows that the recursive algorithm is clearly faster for up to 3 dimensions, for 4 dimensions it is slightly faster, and for 5 and 6 dimensions twice recursive is faster. This is coherent with the theoretical analysis that the interior to boundary ratio and the tall cache requirement become bad for the recursive algorithm, whereas the twice recursive algorithm can in both phases be close to the scanning bound. The anisotropic grids are only reported for the generally faster algorithm.

The heuristic works well, it manages to get within 5 percent of the running time with the best parameters for the recursive algorithm, and within 22 percent for the twice recursive algorithm. In one case the running time of the heuristic is actually reported to be faster than that of the best parameters, which is an artifact of repeating the run and means that the heuristic is optimal up to measurement accuracy. In the following, we will always use the heuristic to chose the parameters.

5.4 Strong Scaling

A classical experiment is that of strong scaling, i.e., comparing the runtime for the same task with different number of threads, depicted in Figure 5. In all cases we see perfect scaling between one and two threads, and in most cases a constant performance for four or more threads, which reflects that the machine has four cores. We also see that two threads already achieve more than half of the best performance.

Table 1 Best runtime of the "Recursive" Algorithm over the searched parameter space and runtime given the parameters chosen by the heuristic. 8 OMP threads on a 4 core CPU (rechenteufel)

$d=2$	Runtime (best) [s]	Parameters (best)	Runtime (heuristic) [s]	Parameters (heuristic)	$\frac{\text{runtime (heuristic)}}{\text{runtime (best)}}$
$\ell=(15,15)$	0.98	`SpawnLevel` $=(13,14)$ `recTile` $=5$	1.00	`SpawnLevel` $=(8,13)$ `recTile` $=5$	**1.01**
$\ell=(20,10)$	0.99	`SpawnLevel` $=(7,8)$ `recTile` $=5$	1.00	`SpawnLevel` $=(3,8)$ `recTile` $=5$	**1.01**
$\ell=(10,20)$	0.88	`SpawnLevel` $=(16,17)$ `recTile` $=7$	0.90	`SpawnLevel` $=(13,18)$ `recTile` $=10$	**1.02**

$d=3$	Runtime (best) [s]	Parameters (best)	Runtime (heuristic) [s]	Parameters (heuristic)	$\frac{\text{runtime (heuristic)}}{\text{runtime (best)}}$
$\ell=(10,10,10)$	1.21	`SpawnLevel` $=(17,20)$ `recTile` $=7$	1.21	`SpawnLevel` $=(13,18)$ `recTile` $=10$	**1.00**
$\ell=(15,8,7)$	1.57	`SpawnLevel` $=(12,14)$ `recTile` $=5$	1.58	`SpawnLevel` $=(8,13)$ `recTile` $=5$	**1.01**
$\ell=(8,7,15)$	1.12	`SpawnLevel` $=(19,20)$ `recTile` $=8$	1.12	`SpawnLevel` $=(15,20)$ `recTile` $=8$	**1.00**

$d=4$	Runtime (best) [s]	Parameters (best)	Runtime (heuristic) [s]	Parameters (heuristic)	$\frac{\text{runtime (heuristic)}}{\text{runtime (best)}}$
$\ell=(8,8,7,7)$	1.77	`SpawnLevel` $=(18,21)$ `recTile` $=6$	1.80	`SpawnLevel` $=(15,20)$ `recTile` $=8$	**1.02**
$\ell=(12,6,6,6)$	2.14	`SpawnLevel` $=(4,17)$ `recTile` $=7$	2.21	`SpawnLevel` $=(11,16)$ `recTile` $=12$	**1.03**
$\ell=(6,6,6,12)$	1.86	`SpawnLevel` $=(15,16)$ `recTile` $=6$	1.98	`SpawnLevel` $=(17,22)$ `recTile` $=6$	**1.06**

d,ℓ	Runtime (best) [s]	Parameters (best)	Runtime (heuristic) [s]	Parameters (heuristic)	$\frac{\text{runtime (heuristic)}}{\text{runtime (best)}}$
$d=5$ $\ell=(6,6,6,6,6)$	2.76	`SpawnLevel` $=(19,20)$ `recTile` $=5$	3.04	`SpawnLevel` $=(17,22)$ `recTile` $=6$	**1.10**
$d=6$ $\ell=(5,5,5,5,5,5)$	3.81	`SpawnLevel` $=(10,22)$ `recTile` $=3$	4.61	`SpawnLevel` $=(18,23)$ `recTile` $=5$	**1.21**

For the 6 dimensional case we observe that the second phase of the twice recursive algorithm has a somewhat unstable performance, and that only for 11 and 12 threads it is close to scanning time. For the 5 dimensional case the performance is stable but the first phase takes two times scanning, reflecting that the subgrids are bigger than they should ideally be. All in all, twice recursive does not quite achieve the possible performance of scanning twice, but it still outperforms the unidirectional scanning bound by a factor of two for six dimension, and almost that for five dimensions.

Table 2 Best runtime of the "Twice Recursive" Algorithm over the searched parameter space and runtime given the parameters chosen by the heuristic.

$d, \boldsymbol{\ell}$	Runtime (best) [s]	Parameters (best)	Runtime (heuristic) [s]	Parameters (heuristic)	$\frac{\text{runtime (heuristic)}}{\text{runtime (best)}}$
$d=2$ $\boldsymbol{\ell}=(15,15)$	1.77	$d_{\text{split}}=2$ `BlockSize` $=8192$	1.85	$d_{\text{split}}=2$ `BlockSize` $=1024$	**1.05**
$d=3$ $\boldsymbol{\ell}=(10,10,10)$	2.12	$d_{\text{split}}=2$ `BlockSize` $=256$	2.90	$d_{\text{split}}=3$ `BlockSize` $=1024$	**1.37**
$d=4$ $\boldsymbol{\ell}=(8,8,7,7)$	2.03	$d_{\text{split}}=3$ `BlockSize` $=1024$	1.99	$d_{\text{split}}=3$ `BlockSize` $=1024$	**0.98**

$d=5$	Runtime (best) [s]	Parameters (best)	Runtime (heuristic) [s]	Parameters (heuristic)	$\frac{\text{runtime (heuristic)}}{\text{runtime (best)}}$
$\boldsymbol{\ell}=(6,6,6,6,6)$	2.42	$d_{\text{split}}=3$ `BlockSize` $=1024$	2.45	$d_{\text{split}}=4$ `BlockSize` $=1024$	**1.01**
$\boldsymbol{\ell}=(10,5,5,5,5)$	1.97	$d_{\text{split}}=3$ `BlockSize` $=2048$	2.22	$d_{\text{split}}=3$ `BlockSize` $=1024$	**1.13**
$\boldsymbol{\ell}=(5,5,5,5,10)$	1.41	$d_{\text{split}}=4$ `BlockSize` $=1024$	1.50	$d_{\text{split}}=4$ `BlockSize` $=1024$	**1.06**

$d=6$	Runtime (best) [s]	Parameters (best)	Runtime (heuristic) [s]	Parameters (heuristic)	$\frac{\text{runtime (heuristic)}}{\text{runtime (best)}}$
$\boldsymbol{\ell}=(5,5,5,5,5,5)$	1.66	$d_{\text{split}}=4$ `BlockSize` $=1024$	1.72	$d_{\text{split}}=4$ `BlockSize` $=1024$	**1.04**
$\boldsymbol{\ell}=(8,5,5,4,4,4)$	2.23	$d_{\text{split}}=4$ `BlockSize` $=8192$	2.30	$d_{\text{split}}=4$ `BlockSize` $=1024$	**1.03**
$\boldsymbol{\ell}=(5,5,4,4,4,8)$	1.55	$d_{\text{split}}=4$ `BlockSize` $=512$	1.89	$d_{\text{split}}=4$ `BlockSize` $=1024$	**1.22**

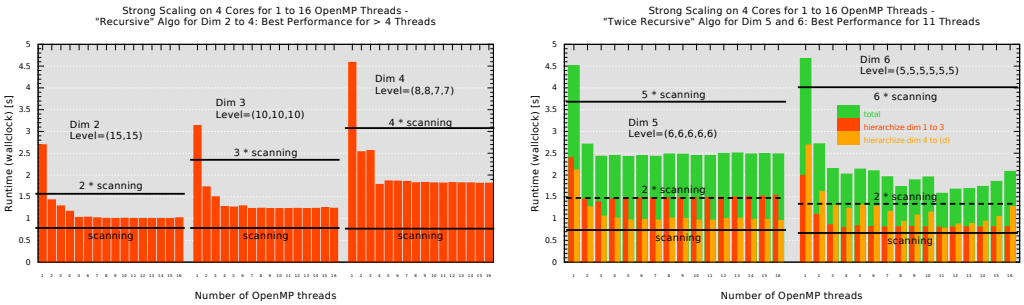

Fig. 5 Scaling of the two algorithms on a four core single CPU machine

5.5 Anisotropic Grids

In the context of the combination technique, many component grids are anisotropic. We address this by considering grids of different dimensions with level sum 30, i.e., roughly 8 GB of data, as reported Figure 6. In all tested cases, the unidirectional bound is beaten, in many cases quite clearly. Three cases achieve almost the best possible performance. The recursive algorithm suffers somewhat from the first dimension being refined further. For the twice recursive algorithm, changing the most refined dimension actually changes the split between the two phases, which has a strong influence on performance. Further, the performance is better if the refined dimension is handled in the second phase.

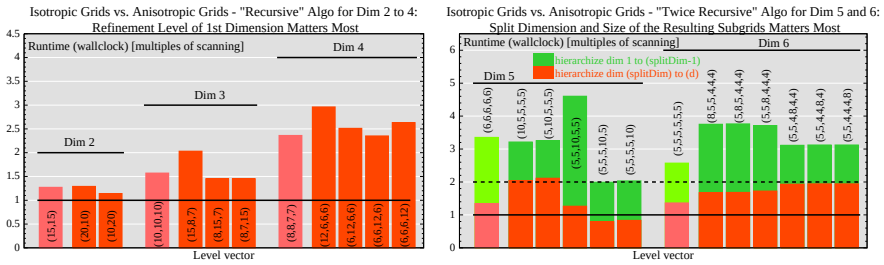

Fig. 6 Influence of anisotropy on the performance

5.6 Speedup over Unidirectional ICCS Code

So far we mainly compared the recursive implementations with the scanning bound, which provides lower bounds, directly for the recursive algorithm, multiplied by 2 for twice recursive and multiplied by d for the unidirectional algorithm. Here we compare this performance with the unidirectional implementation presented in [24]. As we can see in Figure 7, the new implementation is superior for grids with more than a million points, i.e., roughly 100 MB size.

5.7 Increase Grid Size

Another important aspect of the code is how it scales with the size of the grids as depicted in Figure 8. We see that for grids with at least a million points (8 MB size), the performance is stable, and it seems to converge to a constant depending on the dimension. This is in line with the analysis in Section 4.2. In all cases we see the performance well below the bound of the unidirectional algorithm, and for

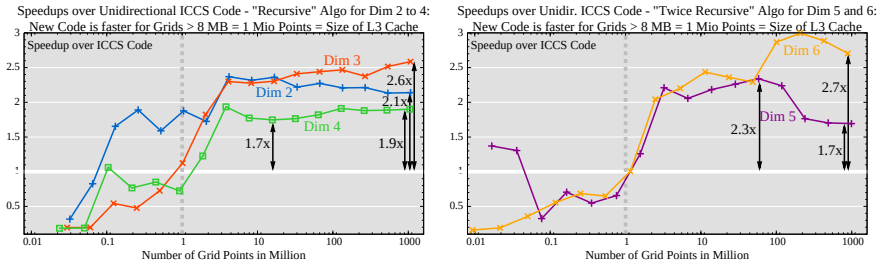

Fig. 7 Comparison of the unidirectional implementation and the recursive ones

dimensions 2 and 3 the recursive algorithm is close to scanning once. In dimensions 5 and 6 the performance is reasonable close to scanning twice, as expected for the twice recursive algorithm.

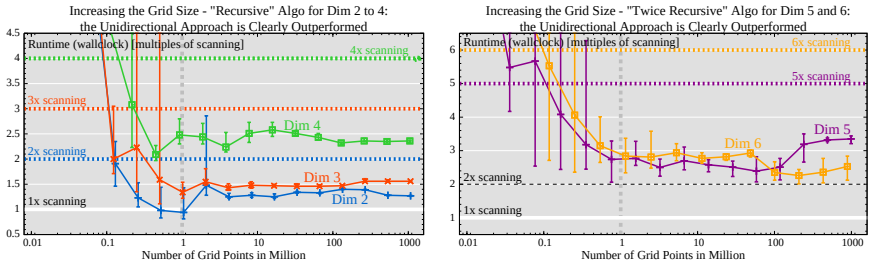

Fig. 8 Performance of the recursive algorithms with respect to grid size. Reports average running time per point (relative to scanning bound) over 10 runs, errorbars show min and max.

5.8 GENE

One important example for the combination technique, as mentioned in the introduction, is the case of using GENE to simulate a fusion reactor, as reported in Figure 9. The peculiar situation here is that the first two dimensions are in phase space and should hence not be hierarchized. This can easily be accommodated by using the twice recursive algorithm with $d_{\text{split}} = 2$. The grid sizes stem from a pilot study performed by Mario Heene and Dirk Pflüger in Stuttgart, and we conducted these experiments on the haswell system (as described earlier). We see that the hierarchization of dimensions 3 to 5 takes less time than scanning the grid twice. These measurements show that also on this architecture the implementation performs well and even the heuristic for choosing the parameters (taken from rechenteufel) is not too system specific.

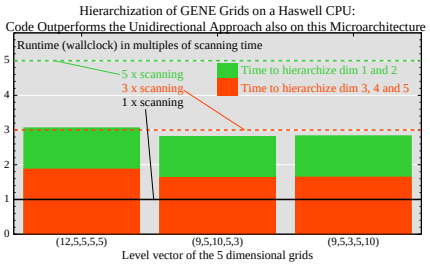

Fig. 9 Grid sizes relevant for the fusion reactor simulation using GENE; haswell system running with 12 threads on 12 cores. The grids have size 32 GB and scanning once is roughly one second.

6 Conclusions

This paper has introduced a novel approach to sparse grid algorithms by deriving a cache-oblivious hierarchization algorithm (Algorithm 2) that avoids the d global phases of the unidirectional principle but applies it recursively to smaller subproblems that fit into cache. For the piecewise linear basis and the component grids of the sparse grid combination technique, a discretization scheme to solve high-dimensional numerical problems, the cache complexity of this algorithm is optimal as the leading term of the cache misses is reduced to scanning complexity. For optimality, Algorithm 2 relies on the tall cache assumption $M = \omega\left(B^d\right)$. The general idea of divide and conquer, however, can also be used to derive hybrid algorithms that merge several but not all phases of the unidirectional principle. These hybrid algorithms trade a weaker tall cache assumption off against a slightly increased complexity. One such algorithm only needs a cache of size $M = \omega(B)$ and, basically, scans the grid twice.

As sparse grids are inherently hierarchical, the divide and conquer approach can also be generalized to other kinds of basis functions and sparse grid tasks such as dehierarchization, i. e., the inverse transformation from the hierarchical basis to the nodal basis, and up-down schemes used to solve PDEs directly in the sparse grid space. In addition, Algorithm 2 is not limited to component grids but can also be applied for adaptive and regular sparse grids. In these cases, the ratio of the number of interior grid points of a grid interval divided by the boundary grid points, which can be seen as progress/costs, becomes worse. As a result, the analysis presented for component grids would need to be altered to show that the leading term of cache misses is also optimal in the setting of regular sparse grids.

The analysis of the I/O complexity of Algorithm 2 is complemented with an implementation. The presented results show, that it is possible to handle additional factors that influence runtime such as vectorization and branch predictions well enough that the memory connection is used almost fully. In particular, the new implementation clearly outperforms previously existing implementations, and in several cases it comes close to optimal performance (as is possible by the memory system).

In the experiments, we see that it is advantageous if the base case is square with respect to cache lines, a case that should be possible to analyze theoretically as well. Then, perhaps, a weaker tall cache assumption might be sufficient.

Acknowledgements We would like to thank Dirk Pflüger and Mario Heene for support and discussions, in particular for enabling the experiments on Hornet. We also thank two anonymous referees for detailed feedback on an earlier draft.

References

1. Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.
2. Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Analysis Applications*, 32(3):866–901, 2011.
3. H.-J. Bungartz and M. Griebel. Sparse grids. *Acta Numerica*, vol. 13:147–269, 2004.
4. Hans-Joachim Bungartz, Alexander Heinecke, Dirk Pflüger, and Stefanie Schraufstetter. Option pricing with a direct adaptive sparse grid approach. *Journal of Computational and Applied Mathematics*, 236(15):3741–3750, 2011. online Okt. 2011.
5. Hans-Joachim Bungartz, Dirk Pflüger, and Stefan Zimmer. Adaptive sparse grid techniques for data mining. In H.G. Bock, E. Kostina, X.P. Hoang, and R. Rannacher, editors, *Modelling, Simulation and Optimization of Complex Processes 2006, Proc. Int. Conf. HPSC, Hanoi, Vietnam*, pages 121–130. Springer-Verlag, 2008.
6. G. Buse, R. Jacob, D. Pflüger, and A. Murarasu. A non-static data layout enhancing parallelism and vectorization in sparse grid algorithms. In *11th International Symposium on Parallel and Distributed Computing, ISPDC 2012, Munich, Germany, June 25-29, 2012. Proceedings*, pages 195–202, Munich, 2012. IEEE.
7. Daniel Butnaru, Dirk Pflüger, and Hans-Joachim Bungartz. Towards high-dimensional computational steering of precomputed simulation data using sparse grids. In *Proceedings of the International Conference on Computational Science (ICCS) 2011*, volume 4 of *Procedia CS*, pages 56–65. Tsukaba, Japan, Springer-Verlag, 2011.
8. P. Butz. Effiziente verteilte Hierarchisierung und Dehierarchisierung auf vollen Gittern. http://d-nb.info/1063333806, 2014. Bachelor's thesis, University of Stuttgart.
9. C Feuersänger. *Sparse Grid Methods for Higher Dimensional Approximation*. PhD thesis, Universität Bonn, 2010.
10. Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *In Proc. 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–297. IEEE Computer Society Press, 1999.
11. J Garcke. *Maschinelles Lernen durch Funktionsrekonstruktion mit verallgemeinerten dünnen Gittern*. PhD thesis, Universität Bonn, 2004.
12. Jochen Garcke and Michael Griebel. On the parallelization of the sparse grid approach for data mining. In Svetozar Margenov, Jerzy Waśniewski, and Plamen Yalamov, editors, *Large-Scale Scientific Computing*, volume 2179 of *Lecture Notes in Computer Science*, pages 22–32. Springer Berlin Heidelberg, 2001.
13. E. Georganas, J. González-Domínguez, E. Solomonik, Y. Zheng, J. Touriño, and K. Yelick. Communication avoiding and overlapping for numerical linear algebra. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 100:1–100:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
14. M. Griebel and H. Harbrecht. On the convergence of the combination technique. In *Sparse grids and Applications*, volume 97 of *Lecture Notes in Computational Science and Engineering*, pages 55–74. Springer, 2014.

15. M. Griebel and W. Huber. Turbulence simulation on sparse grids using the combination method. In N. Satofuka, J. Periaux, and A. Ecer, editors, *Proceedings Parallel Computational Fluid Dynamics, New Algorithms and Applications CFD 94, Kyoto*, pages 75–84, Wiesbaden Braunschweig, 1995. Vieweg.
16. M. Griebel, W. Huber, and C. Zenger. Numerical turbulence simulation on a parallel computer using the combination method. In *Flow simulation on high performance computers II, Notes on Numerical Fluid Mechanics 52*, pages 34–47, 1996.
17. M. Griebel, M. Schneider, and C. Zenger. A combination technique for the solution of sparse grid problems. In *Iterative Methods in Linear Algebra*, pages 263–281. IMACS, Elsevier, 1992.
18. M. Griebel and V. Thurner. The efficient solution of fluid dynamics problems by the combination technique. *Int. J. Num. Meth. for Heat and Fluid Flow*, 5:51–69, 1995.
19. Michael Griebel. The combination technique for the sparse grid solution of PDE's on multiprocessor machines. In *Parallel Processing Letters*, pages 61–70, 1992.
20. B. Harding and M. Hegland. A robust combination technique. In *CTAC-2012*, volume 54 of *ANZIAM J.*, pages C394–C411, 2013.
21. Markus Holtz. *Sparse Grid Quadrature in High Dimensions with Applications in Finance and Insurance.*, volume 77 of *Lecture Notes in Computational Science and Engineering*. Springer, 2011.
22. Jia-Wei Hong and Hsiang-Tsung Kung. I/O complexity: The red-blue pebble game. In *Proceedings of STOC '81*, pages 326–333, New York, NY, USA, 1981. ACM.
23. P. Hupp. *Communication Efficient Algorithms for Numerical Problems on Full and Sparse Grids*. PhD thesis, ETH Zurich, 2014.
24. P. Hupp. Performance of unidirectional hierarchization for component grids virtually maximized. In *2014 International Conference on Computational Science*, volume 29 of *Procedia Computer Science*, pages 2272–2283. Elsevier, 2014.
25. P. Hupp, M. Heene, R. Jacob, and D. Pflüger. Global communication schemes for the numerical solution of high-dimensional PDEs. in preparation., 2014.
26. P. Hupp, R. Jacob, M. Heene, D. Pflüger, and M. Hegland. Global communication schemes for the sparse grid combination technique. In *Parallel Computing - Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 564–573. IOS Press, 2014.
27. Dror Irony, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64(9):1017–1026, 2004.
28. Riko Jacob. Efficient regular sparse grid hierarchization by a dynamic memory layout. In Jochen Garcke and Dirk Pflüger, editors, *Sparse Grids and Applications - Munich 2012*, volume 97 of *Lecture Notes in Computational Science and Engineering*, pages 195–219. Springer International Publishing, 2014.
29. Christoph Kowitz and Markus Hegland. The sparse grid combination technique for computing eigenvalues in linear gyrokinetics. *Procedia Computer Science*, 18(0):449 – 458, 2013. 2013 International Conference on Computational Science.
30. M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. *SIGPLAN Not.*, 26(4):63–74, 1991.
31. A. Maheshwari and N. Zeh. A survey of techniques for designing I/O-efficient algorithms. In Ulrich Meyer, Peter Sanders, and Jop Sibeyn, editors, *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, pages 36–61. Springer Berlin Heidelberg, 2003.
32. A. Murarasu, J. Weidendorfer, G. Buse, D. Butnaru, and D. Pflüger. Compact data structure and scalable algorithms for the sparse grid technique. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP, pages 25–34. ACM, 2011.
33. A. F. Murarasu, G. Buse, D. Pflüger, J. Weidendorfer, and A. Bode. fastsg: A fast routines library for sparse grids. *Procedia CS*, 9:354–363, 2012.
34. C. Pflaum. Convergence of the combination technique for second-order elliptic differential equations. *SIAM Journal on Numerical Analysis*, 34(6):2431–2455, 1997.
35. C. Pflaum and A. Zhou. Error analysis of the combination technique. *Numerische Mathematik*, 84(2):327–350, 1999.

36. D. Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems*. PhD thesis, Institut für Informatik, Technische Universität München, 2010.
37. D. Pflüger, H.-J. Bungartz, M. Griebel, F. Jenko, T. Dannert, M. Heene, A. Parra Hinojosa, C. Kowitz, and P. Zaspel. Exahd: An exa-scalable two-level sparse grid approach for higher-dimensional problems in plasma physics and beyond. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8806 of *Lecture Notes in Computer Science*, pages 565–576. Springer-Verlag, 2014.
38. Harald Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, 1999.
39. Christoph Reisinger. Analysis of linear difference schemes in the sparse grid combination technique. *IMA Journal of Numerical Analysis*, 33(2):544–581, 2013.
40. S. Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. *Soviet Mathematics, Doklady*, 4:240–243, 1963.
41. C. Zenger. Sparse grids. In *Parallel Algorithms for Partial Differential Equations*, volume 31 of *Notes on Numerical Fluid Mechanics*, pages 241–251. Vieweg, 1991.